Lab 7
Modified Instructions

EELE 467
Due date: 10/15/2024

Follow steps 1 through 5 in the textbook. Then follow this document; in other words, don't follow steps 6 through 10 in the textbook.

Every time you reboot your FPGA, the U-boot bootloader programs the FPGA with a *raw binary file* (.rbf). To program the FPGA with your lab 7 bitstream every time the FPGA boots, we can convert the .sof file into an .rbf file.

1. Read section 6.1.4 of the textbook to learn how to convert the .sof into an .rbf file.
2. Place your *soc_system.rbf* on the FAT32 partition of your SD card.

Accessing memory with `/dev/mem`

As stated in the textbook, `/dev/mem` is a character device file that allows access to our system's physical memory. We'll use `/dev/mem` to access our registers on the HPS-to-FPGA lightweight bridge.

In this section, you're going to

1. Test reading and writing registers from Linux using `busybox devmem`.
2. Create a C program that can read/write to/from `/dev/mem`. This program is given to you in this handout, but you will need to understand the code and type it yourself—don't copy/paste!
3. Cross-compile the program for the ARM processor in your SoC FPGA
4. Put your `devmem` program on your SD card so you can run it on the ARM processor.
5. Test reading/writing registers with your `devmem` program.

Remember: the base address of the HPS-to-FPGA lightweight bridge is 0xff200000

busybox devmem

In previous steps of the lab, we've already determined that we can read from and write to our registers via the JTAG-to-Avalon master. Our next step is to verify that we can do the same from Linux using the HPS-to-FPGA lightweight bridge. In the spirit of systematically testing small chunks, we're going to verify that the bridge is connected properly before writing our own software. If we were to write our software and it didn't work, how would we know if the issues was in our software or if we didn't connect the hardware correctly? Fortunately, there are already programs designed for interacting with memory-mapped I/O via `/dev/mem`, so we'll use one of those programs before writing our own.

1. Make sure your SD card is inserted in the DE10 Nano.

2. Connect to your DE10 Nano's UART port using PuTTY (or your favorite serial console program). See section 11.1.1 of the textbook for instructions on setting up PuTTY.
3. Open PuTTY and power on your DE10 Nano.
4. When a login prompt pops up, the login name should be "root". If there is a password, it will also be "root".
5. Check if `busybox devmem` is installed by trying to run `busybox devmem`; if it isn't installed, flash Terasic's Linux Console (kernel 4.5) sd card image to your sd card. I suggest using Balena Etcher to flash the image.
6. Check that you can read from and write to your component's registers.
7. Celebrate! 🎉

`devmem` program

Now that we've verified our HPS-to-FPGA bridge is working, we're going to learn how `/dev/mem` works by writing our own code. Implement the code in the following listing. Be sure to read and understand the code and comments; if you don't understand something, ask!

Any C source code you write and compile needs to be compiled in your Ubuntu VM! That is, do the following sections in your Ubuntu VM.

Put the following source code in `sw/devmem/devmem.c`.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <sys/mman.h> // for mmap
#include <fcntl.h> // for file open flags
#include <unistd.h> // for getting the page size

void usage()
{
    fprintf(stderr, "devmem ADDRESS [VALUE]\n");
    fprintf(stderr, "  devmem can be used to read/write to physical memory via the /dev/mem
    ↪  device.\n");
    fprintf(stderr, "  devmem will only read/write 32-bit values.\n\n");
    fprintf(stderr, "  Arguments:\n");
    fprintf(stderr, "    ADDRESS The address to read/write to/from\n");
    fprintf(stderr, "    VALUE   The optional value to write to ADDRESS; if not given, a read
    ↪  will be performed.\n");
}

int main(int argc, char **argv)
{
```

```
    // This is the size of a page of memory in the system. Typically 4096 bytes.
    const size_t PAGE_SIZE = sysconf(_SC_PAGE_SIZE);

    if (argc == 1)
    {
        // No arguments were given, so print the usage text and exit;
        // NOTE: The first argument is actually the program name, so argv[0]
        // is the program name, argv[1] is the first *real* argument, etc.
        usage();
        return 1;
    }

    // If the VALUE argument was given, we'll perform a write operation.
    bool is_write = (argc == 3) ? true : false;

    const uint32_t ADDRESS = strtoul(argv[1], NULL, 0);

    // Open the /dev/mem file, which is an image of the main system memory.
    // We use synchronous write operations (O_SYNC) to ensure that the value
    // is fully written to the underlying hardware before the write call returns.
    int fd = open("/dev/mem", O_RDWR | O_SYNC);
    if (fd == -1)
    {
        fprintf(stderr, "failed to open /dev/mem.\n");
        return 1;
    }

    // mmap needs to map memory at page boundaries; that is, the address we are
    // mapping needs to be page-aligned. The ~(PAGE_SIZE - 1) bitmask returns
    // the closest page-aligned address that contains ADDRESS in the page.
    // For a page size of 4096 bytes, (PAGE_SIZE - 1) = 0xFFF; extending this
    // to 32-bits and flipping the bits results in a mask of 0xFFFF_F000.
    // AND'ing with this bitmask forces the last 3 nibbles of ADDRESS to be 0,
    // which ensures that the returned address is a multiple of the page size
    // (4096 = 0x1000, so indeed, any address that is a multiple of 4096 will
    // have the last 3 nibbles equal to 0).
    uint32_t page_aligned_addr = ADDRESS & ~(PAGE_SIZE - 1);
    printf("memory addresses:\n");
    printf("-----------------------------------------------------------------------\n");
    printf("page aligned address = 0x%x\n", page_aligned_addr);

    // Map a page of physical memory into virtual memory. See the mmap man page
    // for more info: https://www.man7.org/linux/man-pages/man2/mmap.2.html.
    uint32_t *page_virtual_addr = (uint32_t *)mmap(NULL, PAGE_SIZE,
        PROT_READ | PROT_WRITE, MAP_SHARED, fd, page_aligned_addr);
    if (page_virtual_addr == MAP_FAILED)
    {
        fprintf(stderr, "failed to map memory.\n");
```

```
69          return 1;
70      }
71      printf("page_virtual_addr = %p\n", page_virtual_addr);
72
73      // The address we want to access might not be page-aligned. Since we mapped
74      // a page-aligned address, we need our target address' offset from the
75      // page boundary. Using this offset, we can compute the virtual address
76      // corresponding to our physical target address (ADDRESS).
77      uint32_t offset_in_page = ADDRESS & (PAGE_SIZE - 1);
78      printf("offset in page = 0x%x\n", offset_in_page);
79
80      // Compute the virtual address corresponding to ADDRESS. Because
81      // page_virtual_addr and target_virtual_addr are both uint32_t pointers,
82      // pointer addition multiplies the pointer address by the number of bytes
83      // needed to store a uint32_t (4 bytes); e.g., 0x10 + 4 = 0x20, not 0x14.
84      // Consequently, we need to divide offset_in_page by 4 bytes to make the
85      // pointer addition return our desired address (0x14 in the example).
86      // We use volatile because the value at target_virtual_addr could change
87      // outside of our program; the address refers to memory-mapped I/O
88      // that could be changed by hardware. volatile tells the compiler to
89      // not optimize accesses to this memory address.
90      volatile uint32_t *target_virtual_addr = page_virtual_addr + offset_in_page/sizeof(uint32_t
        ↪  *);
91      printf("target_virtual_addr = %p\n", target_virtual_addr);
92      printf("-----------------------------------------------------------------\n");
93
94      if (is_write)
95      {
96          const uint32_t VALUE = strtoul(argv[2], NULL, 0);
97          *target_virtual_addr = VALUE;
98      }
99      else
100     {
101         printf("\nvalue at 0x%x = 0x%x\n", ADDRESS, *target_virtual_addr);
102     }
103
104     return 0;
105 }
```

Cross compiling

Read textbook sections 11.1.4.1 and 11.1.4.2 to learn how to cross-compile your code for the ARM processor. Ignore any mention of the "NFS server", as we don't have that set up yet.

In short:

1. Install the cross-compiler toolchain:

```
sudo apt install gcc-arm-linux-gnueabihf
```

2. Cross-compile:

```
arm-linux-gnueabihf-gcc -o devmem -Wall -static devmem.c
```

We need to the use `-static` flag to force *static linking* of libraries instead of *dynamic linking*. The Linux version on your SoC FPGA is very old and thus has an ancient version of GCC and the GNU standard C library (glibc). The version of gcc/glibc on your Ubuntu VM is much newer. When using dynamic linking, the executable searches for library code in *shared object (.so) files*. There is an incompability between glibc versions that causes dynamic linking to fail when running on the SoC. Using static linking forces the used libraries to be bundled in the executable.

Put `devmem` in ARM rootfs

Once you've compiled your `devmem` source code, you'll put it on your SD card. Your SD card has two primary partitions: a FAT32 partition for bootloader-related files, and an EXT4 partition for the Linux filesystem. You'll need to put your `devmem` executable in `/home/root/` of the Linux filesystem.

1. Power off your SoC FPGA.
2. Plug your SD card into your computer.
3. Open the SD card in your VM. You have to open it in the Ubuntu VM because Windows can't read the Linux EXT4 filesystem.
4. Using `sudo`, or by launching a file browser with admin privileges, copy `devmem` to `/home/root` on your SD card.

Test your `devmem` program

Put your SD card back in the DE10 Nano and boot your board. Once you login, test your `devmem` program to make sure it works. If it doesn't work, have fun debugging!  You'll have to edit the file, cross-compile it, copy the executable to the SD card, and repeat until you've fixed all the bugs. Alternatively, you can edit your source directly on the SoC FPGA and compile it natively in your PuTTY terminal.

Deliverables

Demonstration

1. In **System Console**, put your system in *software control mode* and write a value to the LEDs.
2. In **System Console**, put your system in *hardware control mode* and set your `base_period` to 0.125 seconds.
3. In **Linux, using `devmem`**, put your system in *software control mode* and write a value to the LEDs.
4. In **Linux, using `devmem`**, put your system in *hardware control mode* and set your `base_period` to 0.5625 seconds.

GitHub Submission

Follow the workflow given in your repository (`docs/workflow.md`), per usual. Make sure all your files are committed.

In your `docs/lab7.md` file, answer the following questions:

1. What hex value did you write to `base_period` to have a 0.125 second base period?
2. What hex value did you write to `base_period` to have a 0.5625 second base period?